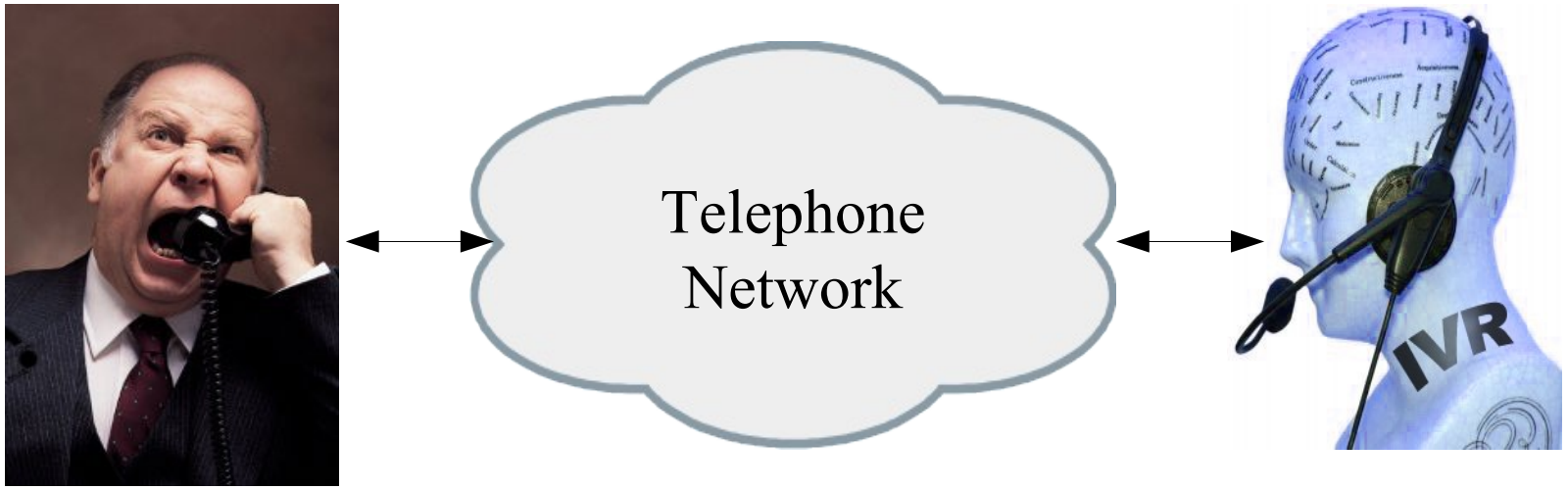
Asynchronous Zero-copy API for Embedded IVR Application

Alexey Volkov

ELC 2010

What is IVR



Interactive **V**oice **R**esponse

A telephony technology which allows us to interact with the remote information system using the telephone

IVR Applications

- Automatic attendants
- Customer self-service
- Prepaid cards activation
- Voice mail
- Voice dialing

IVR Implementations

- **Dedicated**
  - **PC-based**
  - **Board-based**
- **Embedded**
  - **Implemented on switching node**

Interaction logic

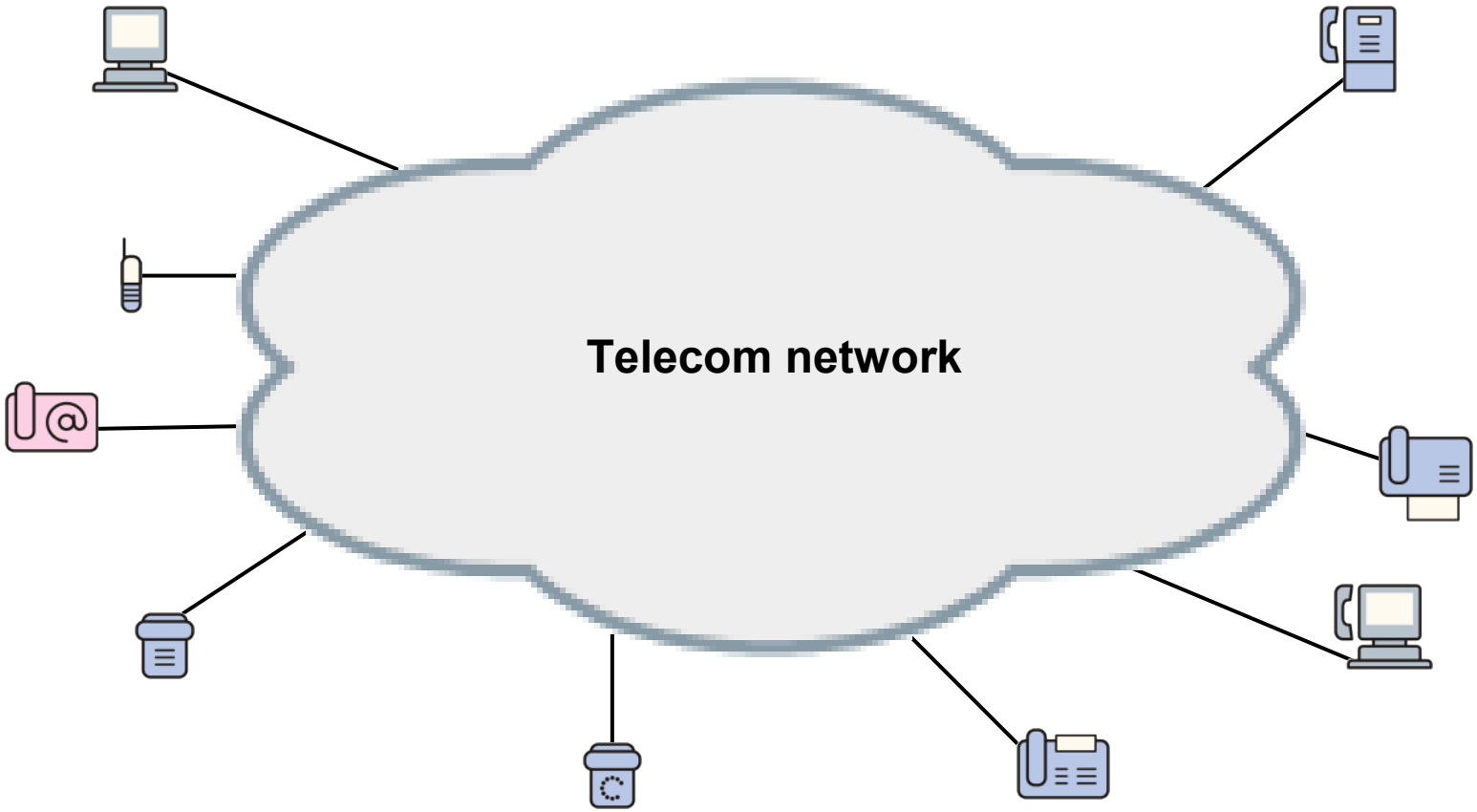
- **Hard-coded**
  - **A few scenarios or modes implemented in software**
- **Configurable**
  - **More complex scenarios stored in config files or DB**
- **Scripts**

Voice XML

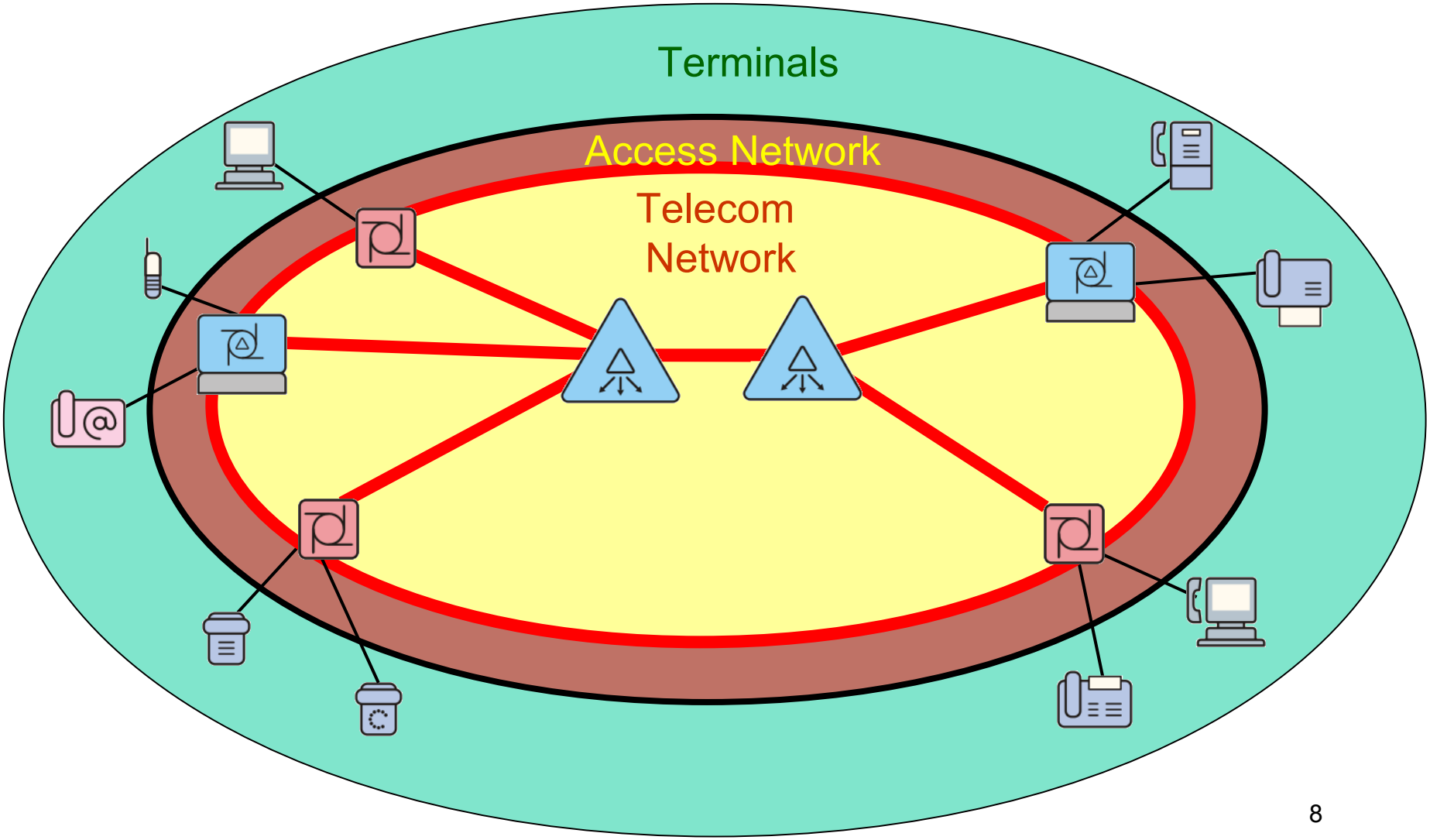
```
<?xml version="1.0" encoding="UTF-8"?>
<vxml version = "2.1" >
  <form>
    <block>
    <prompt>
      Hello World
    </prompt>
    </block>
  </form>
</vxml>
```

http://www.w3.org/TR/voicexml20/

NGN Network



Network Elements



IVR-enabled Network Elements

- **Integrated Call Server**
  - **Access Node + TDM Switch + VoIP GW**

- **Signaling and Media Gateway**
  - **TDM Switch + VoIP GW**

MGW provides transformation of TDM signaling (SSN7, DSS1 and V5.2) into IP-based session-control protocols (M2UA, M3UA, M2PA, IUA and V5UA)

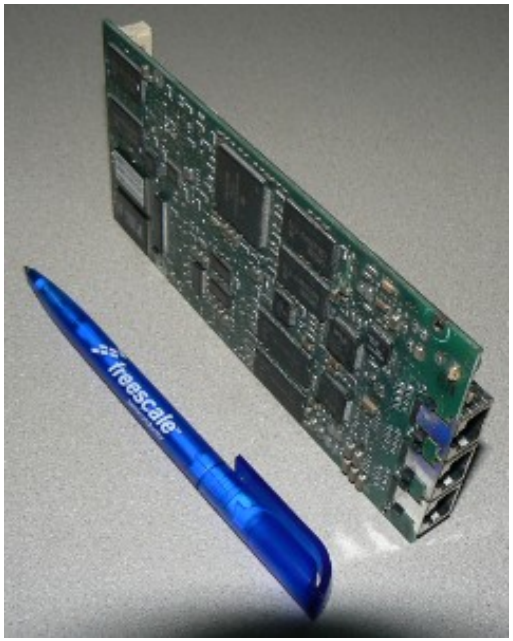
Why?

- Why do we bother with TDM?
  - Strategy: we offer evolutionary path from TDM to NGN
- Why do we run IVR on MGW?
  - Connectivity to both worlds:
    - Native to PSTN
    - IP subscribers interconnected over on-board VoIP DSPs
  - Marketing: buy a TDM-to-IP box, get IVR platform for free
  - We can handle it here.

MGW: Carrier Board

- provides basic infrastructure and interfaces (TDM, PCI, ETH) for plug-in PMC processor boards
- accepts up to four daughter cards (1 MCU and up to 3 optional PCUs)

Processor Board

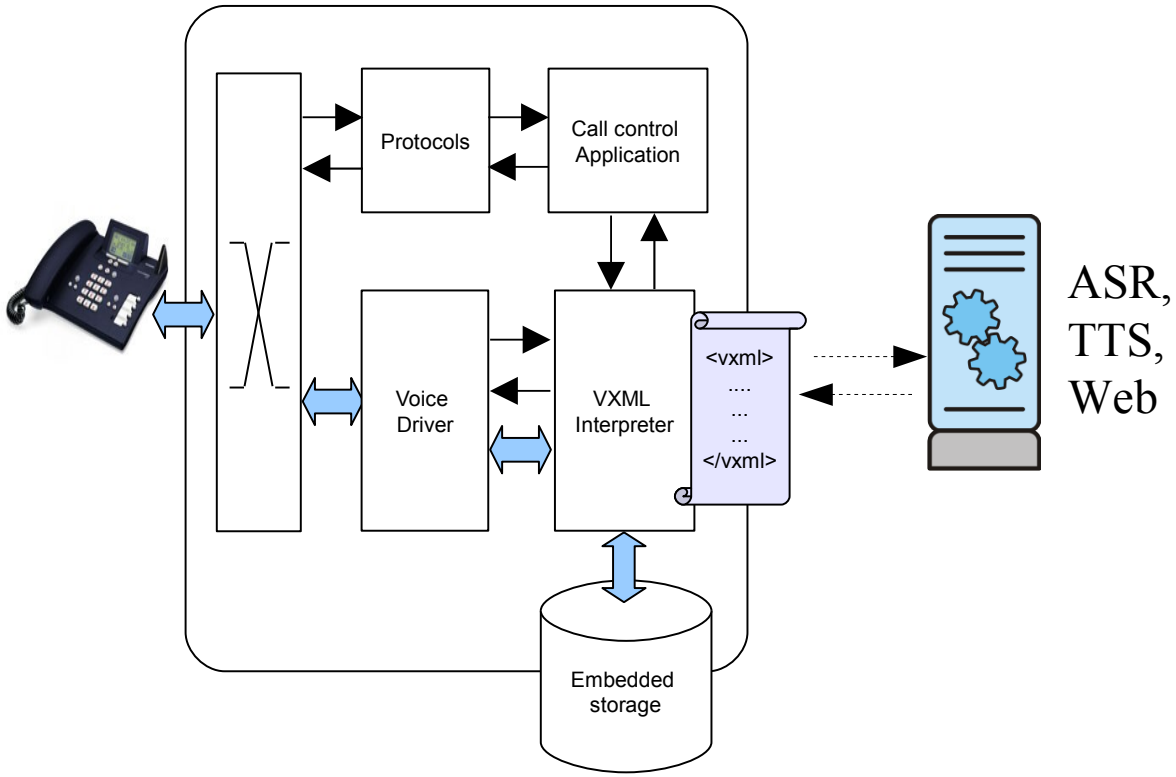


- Standard (PMC) format
- Same for MCU and PCU
- MPC8260 (Motorola PowerQUICC II)
  - 300 MHz core
  - CPM Co Processor
- 128MB SDRAM, 2MB SRAM, 8MB FLASH
- 32-bit, 33MHz PCI (3.3V or 5V)
- 2x 100BaseT Ethernet
- 1x 10Mbps AUI or UTP Ethernet
- 2x RS-232 serial interfaces

MGW: Processor boards

- **MCU**
  - **Controls all units of carrier board (disk, TDM switch, framers, system registers, ethernet switch)**
  - **Loads OS images to PCU boards,**
  - **TDM signaling and connection handling,**
  - **Call control**
- **PCU**
  - **Controls DSPs**
  - **And nothing more.**

IVR Subsystem Architecture

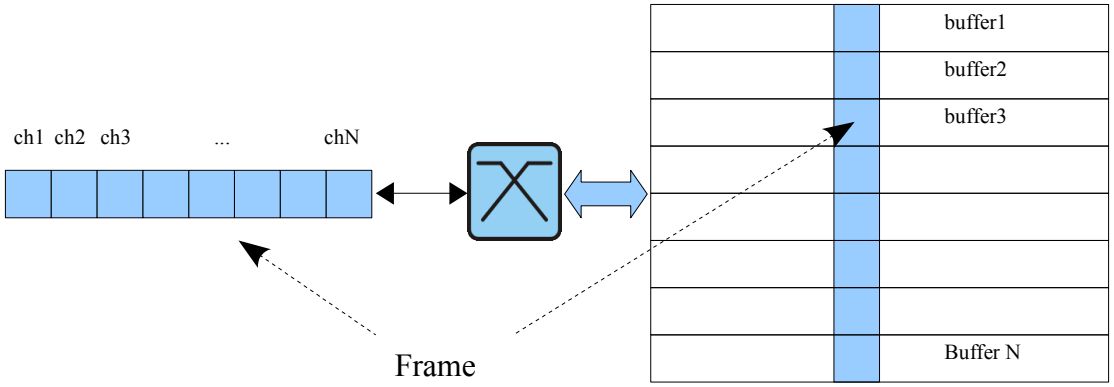


IVR Subsystem

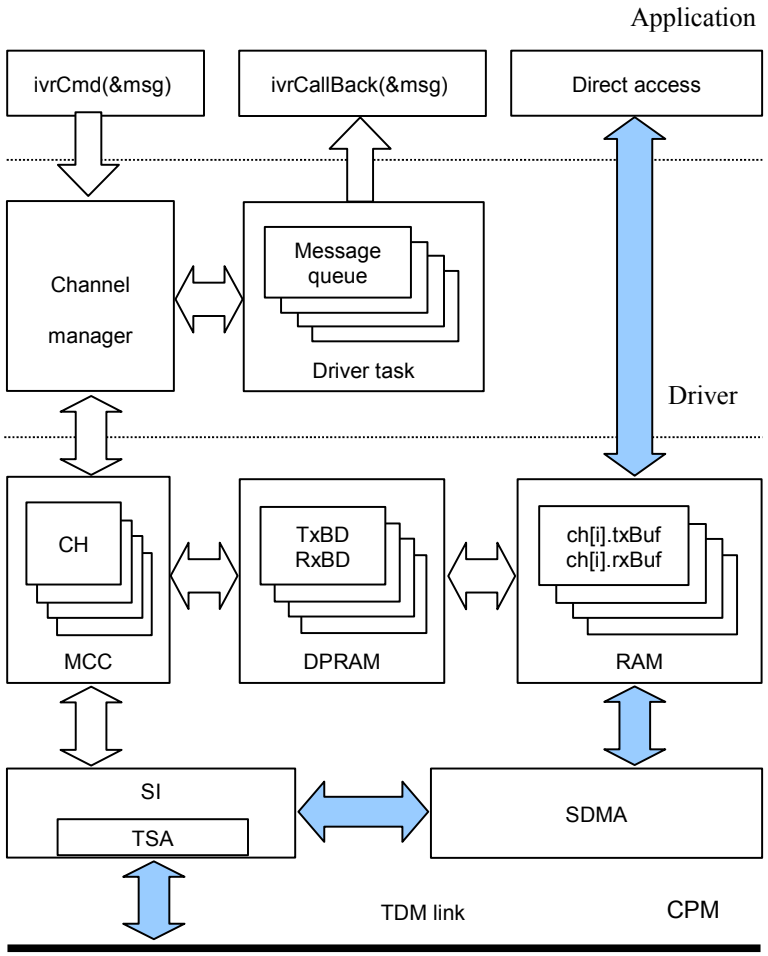
- **Call Control:**
  - Activates VXML Interpreter for inbound calls to special prefixes
  - Connects caller to corresponding IVR channel
- **VXML Interpreter:**
  - Loads and parses VXML script
  - Acquires voice prompts from local or ext. storage
  - Handles user input (captures speech and DTMF tones)
- **Voice Driver**
  - Performs streaming of raw pcm voice data to/from TDM line

TDM-specific features

- Constant baud rate 64 kbit/s (B-channel)
- No buffering on terminal: user already hears what we put on line
- TDM line doesn't sleep: when you hear silence, the idle code is transmitted to your timeslot



Legacy IVR Design (VxWorks)



- driver and application can share the data, and the buffer pointer is present in every driver message

- application and driver can directly call each other's functions

- We can assume MCC as 128 parallel DMA engines. We must be in time to update their buffers

Voice Driver Interface

```
/* Driver message */
typedef struct {
  ushort  link;          /* TDM link */
  ushort  channel;       /* channel number */
  ushort  buffer;        /* pointer to data buffer */
  ushort  bufferSize;    /* block size */
  ushort  msgType;       /* command or response */
} t_ivrDrvMsg;

/* Driver command */
int ivrCmd(t_ivrDrvMsg *msg);

/* Application callback */
int ivrCallBack(t_ivrDrvMsg *msg);
```

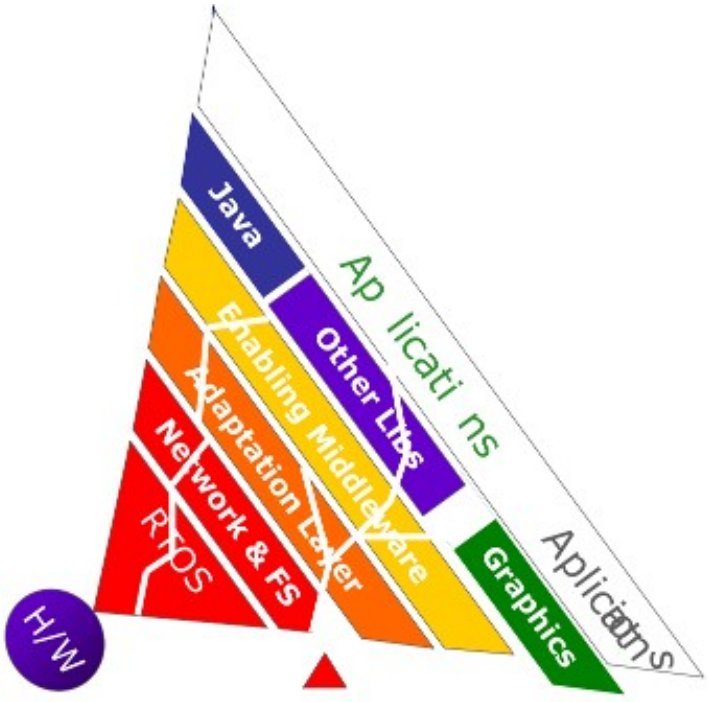
Commands and Responses

```
/* Commands */
typedef enum {
 IVR_INIT_CHANNEL = 0x100,
 IVR_PLAY_BLOCK,
 IVR_RECORD_BLOCK,
 IVR_STOP_PLAY,
 IVR_STOP_RECORD,
 IVR_RELEASE_CHANNEL
} t_ivrDrvCmd;
```

```
/* Responses */
typedef enum {
 IVR_PLAY_GET_NEXT_BLOCK =
0x200,
 IVR_BLOCK_RECORDED,
 IVR_PLAY_END_OF_DATA,
 IVR_CHANNEL_RELEASED
} t_ivrDrvResponse;
```

Why do we go to Linux

We have:



VxWorks
software stack

We need:

- Unified platform for our product line,
- Lower cost of development,
- Safety,
- Shared experience of community

Challenges for redesign

- We should make Voice Driver a real device driver, not a library
- No kernel-to-user callbacks allowed: we should find asynchronous notification facility
- No shared pointers user-to-kernel allowed, but we must support up to 256 channels on PQIII with no performance drop on L1: API must be Zero-Copy
- We must keep the Interface if possible

Porting legacy BSP library

- **Fix up BSP library:**
  - **Use ioremap() to access CPM registers**
  - **Use virt_to_phys() to convert addresses assigned to Buffer Descriptors**
  - **Find appropriate replacement for semTake/semGive**
  - **msgQSend/msgQReceive ?**

Registering IVR device

- **Register «/dev/ivr» device**

```
tatic const struct file_operations ivrDevice_fops= {
        .open   =ivrDeviceOpen,
        .release=ivrDeviceRelease,
        .read    =ivrDeviceRead,
        ...
};

static struct miscdevice ivr_dev = {
        “ivrDrv”,
        ...
        &ivrDevice_fops
};

int __init initIvrDriver(void)
{
        /* Init Motorola BSB library*/
        motSwInit();
        ...
        /* Register driver in Linux */
        misc_register(&ivr_dev);
        return 0;
}
```

Choosing API

- **Blocking read()/write()?**
- **tty_driver?**
- **Netlink sockets?**
- **ioctl()?**
  - **Can start an operation and return immediately**
  - **We can keep old interface**

```
int ivrDeviceIoctl(struct inode *inode, struct file *file, uint command,
                                ulong parameters)
{
        ...
        t_ivrDrvMsg cmd;
        prepare_cmd (&cmd, command);
        ivrCmd(&cmd); /*  pass the command to legacy API  */
        ...
}
```

Shared memory access

- get_user_pages()?
- vmsplice()?
- mmap()?
  - Can be used moth for TX and RX paths
  - Can be hidden in channel initialization (*do_mmap()*)
  - * Application no longer in charge of memory allocation for TX

Events handling

- **select()?**
- **poll()?**
- **epoll()?**
  - **O(1) complexity, overhead does not depend on number of connections or events**

All of them require a *descriptor* for polling.

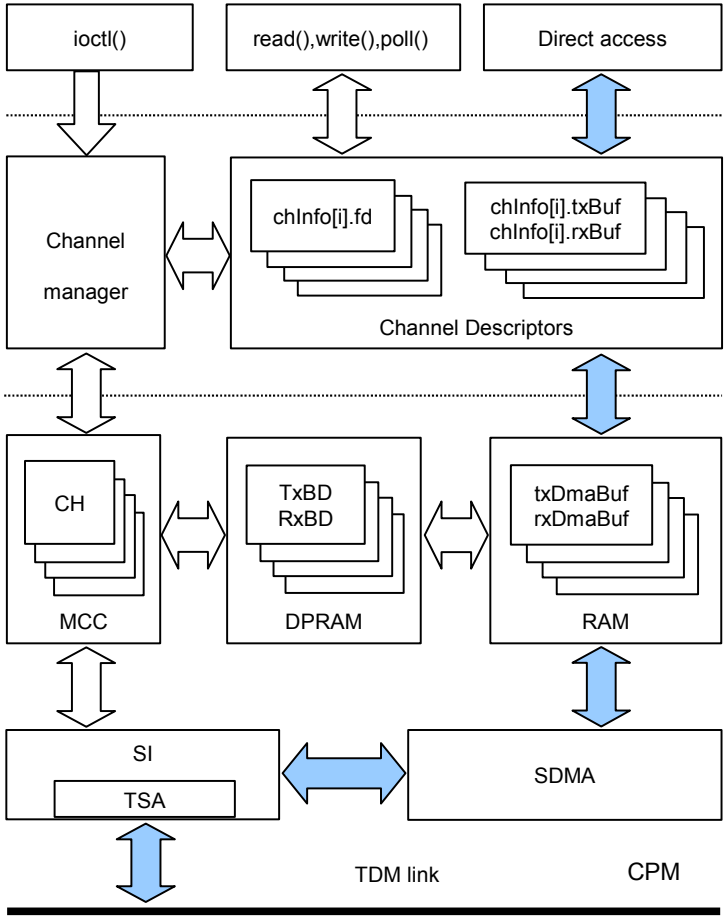
Event notification

- kevent?
- rt-signals?
- eventfd()?
  - Can be used as an event wait/notify mechanism by the kernel to notify user-space applications of arbitrary events.

Solution concepts

- **We should implement «channel descriptor» on basis of *eventfd()* with additional functionality:**
  - **provides zero-copy access to channel buffers;**
  - **accepts commands and forward them to underlying L1 function calls;**
  - **notify the application of channel events (readiness, errors, etc).**

New design



Channel descriptor Implements:

- **read()**
- **write()**
- **mmap() (called implicitly)**
- **poll()**

Channel initialization

- **Added channel_fd creation, memory allocation and mapping:**

```
....

/* Create channel descriptor  */
get_channel_fd(pchannel);

/* Allocate consistent DMA-capable block for RX / TX buffers  */
pchannel->chBuff = (uchar *) dma_alloc_coherent(NULL,
                        chBuffSize, &pchannel->dmaBaseAddr, GFP_KERNEL);
pchannel->userBaseAddr = (uchar *) do_mmap (pchannel->channel_file, 0,  chBuffSize,
PROT_READ |                              PROT_WRITE ,  MAP_SHARED, 0);
/* return channel pointer to user  */
....
```

Creating channel descriptor

```
/* Channel descriptor's control structure */
struct channel_fdctx {
        wait_queue_head_t wqh;          /* wait queue */
        int count;                      /* event counter */
        ivr_channel_t  *pchannel;       /* channel       */
        t_ivrChannelMsg chCmd;          /* command */
        t_ivrChannelMsg chResponse;     /* response  */
};

/* Creating a channel descriptor  */
int create_channel_fd(ivr_channel_t * pchannel)
{
struct channel_fdctx *ctx;
struct inode *inode;
ctx = kmalloc(sizeof(*ctx), GFP_KERNEL);
init_waitqueue_head(&ctx->wqh);
ctx->count = 0;
ctx->channel = pchannel;
anon_inode_getfd(&pchannel->channel_fd ,
        &inode, &pchannel->channel_file,  "[channelfd]",
                &channel_fdfops, ctx);
return pchannel->channel_fd;
}
```

Write

- **write() sends a command to a channel:**

```
static ssize_t channel_fdwrite(struct file *file, const char __user *buf, size_t
count,  loff_t *ppos)
{
        struct channel_fdctx *ctx = file->private_data;
        /*copy command from user space*/
        spin_lock_irq(&ctx->wqh.lock);
        copy_from_user(&ctx->chCmd, buf,
sizeof(t_ivrChivrelMsg)))
      /* execute command - call legacy API  */
        int res = ivrCmd(ctx->pchivrel, &ctx->chCmd);
        spin_unlock_irq(&ctx->wqh.lock);
        return res;
}
```

Read

- **read() returns a response, when awakened**

```
static ssize_t channel_fdread(struct file *file, char __user *buf, size_t
count,  loff_t *ppos)
{
        struct channel_fdctx *ctx = file->private_data;
        ssize_t res;
        spin_lock_irq(&ctx->wqh.lock);
        if (ctx->count) {
                ctx->count = 0;
                if (waitqueue_active(&ctx->wqh))
                        wake_up_locked(&ctx->wqh);
        }
        spin_unlock_irq(&ctx->wqh.lock);
       copy_to_user((void *) buf, &ctx->chResponse,
sizeof(t_ivrDrvMsg)));
        return res;
}
```

Polling

- **This code is invoked when user waits on poll()**

```
static unsigned int channel_fdpoll(struct file *file, poll_table
*wait)
{
        struct channel_fdctx *ctx = file->private_data;
        unsigned long flags;
        poll_wait(file, &ctx->wqh, wait);
        spin_lock_irqsave(&ctx->wqh.lock, flags);
        if (ctx->count > 0 )
                events  |= POLLIN | POLLOUT;
        spin_unlock_irqrestore(&ctx->wqh.lock, flags);
        return events;
}
```

Firing Events

- **This code is invoked by ISR:**

```
int channel_fd_event(struct file *file,
                                t_ivrChannelMsg *msgResponse)
{
spin_lock_irqsave(&ctx->wqh.lock, flags);
/* Save response, increase the counter and wake up the user */
...
ctx->count++;
if (waitqueue_active(&ctx->wqh))
        wake_up_locked(&ctx->wqh);
spin_unlock_irqrestore(&ctx->wqh.lock, flags);
return 0;
}
```

Memory mapping

```
struct vm_operations_struct ivr_drv_vma_ops =
{
  .open = ivr_drv_vma_open,     /* open            */
  .close = ivr_drv_vma_close,   /* close           */
  .nopage = ivr_drv_vma_nopage  /* nopage          */
};

static int channel_fdmmap(struct file *fp,
                          struct vm_area_struct *ivr_drv_vma)
{
ivr_drv_vma->vm_ops = &ivr_drv_vma_ops;
ivr_drv_vma->vm_flags |= VM_RESERVED;
ivr_drv_vma_open(ivr_drv_vma);
return OK;
} /* end of channel_fdmmap() */
```

open(): get_page() for each page of the buffer,
close(): put_page() for each page of the buffer
nopage(): returns virt_to_page(__va(phyaddr));

Programming example

```
int ivrExample(void)
{
 struct epoll_event ev;
 fdIvrDrv=open("/dev/ivrDrv", 0);
 /* Create epoll queue */
 epfd = epoll_create(IVR_QUEUE_LEN);
 /* Init all the channels */
 for (i = 0; i < NUMBER_OF_CHANNELS; ++i)
 {
            chInfo[i].link = IVR_TDM_LINK; /* TDM link for IVR*/
           chInfo[i].channel = i; /* channel number */
           /* Initialize the channel */
           retCode=ioctl(fdAnnDrv, IVR_INIT_CHANNEL, (unsigned int) &chInfo[i] );
           ev.events = EPOLLIN | EPOLLOUT | EPOLLERR;
           /* store channel info pointer in event structure */
            ev.data.ptr = chInfo[i]
           /* add channel file descriptor to event queue */
           retCode=epoll_ctl(epfd, EPOLL_CTL_ADD, chInfo[i].channel_fd, &ev);
 } /* for */
...
 / /That's how the commands are invoked:
 /* chInfoPtr->txBuff  should already contain the data we want to play */
 chMsg.bufferOffset = 0;  /* offset from starting pointer of txBuff */
 chMsg.bufferSize = 8000; /* chunk size */
 chMsg.msgType = IVR_PLAY_BLOCK; /* or whatever command */
 /* Send command to channel */
 write(chInfoPtr->channel_fd, ( void *) &chMsg, sizeof(chMsg) );
}
```

Handling events

Handling of channel events is based on the same algorithm as classical socket-based connection handling:

```
void ivrMsgHandlingTask(void)
{
struct epoll_event ivr_events[NUM_OF_EVENTS];
struct epoll_event *events = &ivr_events[0];
for ( ; ; ){
 /* wait on queue - 2 seconds timeout */
int  nfds = epoll_wait(epfd, events, NUM_OF_EVENTS, 2000);
 /* For all events fired at a moment */
 for(i = 0; i < nfds; ++i)  {
   /* Get channel description pointer from event */
   channel_fd = events[i].data.fd;
   /* Read the message from channel file descriptor */
   read(channel_fd, ( void *) &chMsg,  sizeof(chMsg));
   /* Handle the message */
   ivrCallBack(&chMsg);
} /* for */
}
```

Performance

- **Loop-back test application based on algorithm from the example above shows 4-5% CPU consumption on 128 channels, what is not worse then on VxWorks.**

Summary

- Presented solution allows us to move rather complex system to Linux OS without significant changes in logic of interactions,
- IPC implementation avoids performance loss on channel layer.

Questions?

volkov@iskratel.si

References

• Interactive Voice Response, http://en.wikipedia.org/wiki/Interactive_voice_response
• Voice Extensible Markup Language (VoiceXML) Version 2.0, W3C Recommendation, 16 March 2004, http://www.w3.org/TR/voicexml20/
• MPC8560 Reference Manual, Rev. 1, Freescale Semiconductors, 2004
• Migrating legacy VxWorks applications to Linux, white paper by Bill Weinberg, Linux Pundit for MontaVista Software, October 22, 2008
• The need of Asynchronous, Zero-copy Network API, Ulrich Drepper, Red Hat, http://people.redhat.com/drepper/newni.pdf
• Improving (network) I/O performance, Davide Libenzi, 2006 http://www.xmailserver.org/linux-patches/nio-improve.html
• Using epoll() for Asynchronous Network Programming, Alexey Kovyrin, 2006 http://blog.kovyrin.net/2006/04/13/epoll-asynchronous-network-programming/lang/en/